CSE 505, Fall 2005, Assignment 2
Due: Friday 28 October 2005, 4:30PM

`hw2.tar`, available on the course website, contains several Caml files you will need.
Last updated: October 15

1. **Semantics for Regular Expressions**
   Here is a grammar for regular expressions (r) and strings (s), where x ranges over *characters*:

$$\begin{array}{rcl} r & ::= & x \mid r \cdot r \mid r \vee r \mid r^* \\ s & ::= & \epsilon \mid x \mid ss \end{array}$$

   ϵ is the zero-character string. A string of the form $s_1 s_2$ is s_1 concatenated with s_2. We allow "implicit reassociation" meaning a string like `xyz` can "instantiate" $s_1 s_2$ 4 ways (with ϵ for s_1, `x` for s_1, `xy` for s_1, or `xyz` for s_1).

   Here is a semantics in English: A program is a pair (r, s) that evaluates to "true" if "r matches s" and "false" otherwise. These are the only ways r can match s:

   - For any character x, the regular expression x matches the string x.
   - $r_1 \cdot r_2$ matches $s_1 s_2$ if r_1 matches s_1 and r_2 matches s_2.
   - $r_1 \vee r_2$ matches s if either r_1 or r_2 matches s.
   - r_1^* matches s if $s = \epsilon$ or ($s = s_1 s_2$, r_1 matches s_1, and r_1^* matches s_2).

   (a) Give a formal, large-step semantics for regular expressions. Make your judgment of the form $s \in r$ (meaning s is in the language or r, i.e., r matches s). There should be a derivation of $s \in r$ if and only if (r, s) evaluates to "true" in the sense described above. Your solution should be a *direct translation* of the English above. You need 6 inference rules.

   (b) Implement an interpreter in Caml for regular expressions. (We have written the type definition and parser for you; see the provided files.) Your interpreter should be a *direct translation* of your formal semantics. Hints:
   - Caml does not have "implicit reassociation" of strings, so write a recursive function that "splits" a string into all possible pairs of substrings: `string -> (string * string) list`.
   - The sample solution is 18 lines (including the function above) thanks to library functions `String.length`, `Str.string_before`, `Str.string_after`, and `List.exists`.

   (c) Your interpreter may not terminate! Give two examples of an (r, s) for which the interpreter loops infinitely. Have one example be such that r matches s and one such that r does not match s.

   (d) Make a small change such that your interpreter always terminates. Hint: there's only one place where you need to "skip" one particular string-split when making a recursive call.

   (e) Provide a modified formal semantics that corresponds to the change you made to the interpreter. Hint: Change only one rule. It's fine to say "and use all the other rules from part (a)."

   (f) Prove that the "part (a) semantics" and "part (e) semantics" are equivalent, i.e., there is a derivation of $s \in r$ in part (a) *if and only if* there is a derivation of $s \in r$ in part (e).

   (g) Give a "pseudo-denotational" semantics of regular expressions by translating them into ML functions of type `string->bool`. The result of your translation should not use the regular-expression type in any way and should always terminate when passed a string.

2. **Locks and Simulating Non-Pre-Emption**
We continue our investigation of IMP with threads from homework 1. We can define a version with *optional yield* as follows:

- Start with the solution to problem 2(a) on homework 1.
- Change the rule:

$$\frac{}{H \ ; \ \mathsf{yield} \ ; \ \cdot \rightarrow H \ ; \ \mathsf{skip} \ ; \ \cdot \ ; \ \mathsf{false}} \ \textsc{yield1}$$

to:

$$\frac{}{H \ ; \ \mathsf{yield} \ ; \ q \rightarrow H \ ; \ \mathsf{skip} \ ; \ q \ ; \ \mathsf{false}} \ \textsc{yield1}$$

Part (b) below involves this "optional-yield, non-preemptive language".

(a) Define a formal small-step operational semantics for IMP with *pre-emptive scheduling* (the current thread can change at any step) and *locks*:

- A lock is just an integer c.
- A program state is $H; s; q; L$ where L is a set of locks (the locks currently "held"). ($L ::= \emptyset \mid L \cup \{c\}$)
- Inference rules have the form $H; s; q; L \rightarrow H'; s'; q'; L'; b$ (like in HW1 but with lock-sets).
- There is no yield.
- There are two new statement forms, acquire(c) and release(c):
- If c is not in the current "held-lock" set, then acquire(c) can add it to the current "held-lock" set. (Otherwise, acquire(c) "cannot execute" (i.e, there will be no derivation other than to pre-empt the thread).
- If c is in the current "held-lock" set, then release(c) can remove it from the current "held-lock" set. (Otherwise, release(c) "cannot execute" (i.e, there will be no derivation other than to pre-empt the thread).

(b) Define a (meta)function *translate* (on paper) that takes a statement from the "optional yield, non-preemptive language" and produces a statement in the "pre-emptive with locks" language. Your function should be *meaning-preserving*: The input and output of the translation should be statements that can produce all the same terminal states (ignoring the final "held-lock" set in the latter). Hints:

- Use one (meta)function for the "whole program" and for the substatement of a spawn. Use a helper function for recursing on other smaller statements. So your two functions will be mutually recursive.
- Use one lock (e.g., 0). Getting pre-empted "is okay" if a thread holds a lock that every other thread tries to acquire before "doing anything". Of course, if a thread never releases "the lock" no other will ever "do anything".

(c) Explain carefully but informally why we made yield optional for the source language of our translation.

3. **Extra Credit**

   (a) In Caml, define a finite-state-machine type, define a translation from `Ast.regexp` to this type, and implement the regular-expression language by seeing if the finite-state-machine accepts the string. (Find a textbook that describes the regular-expression to finite-state-machine translation. Do not use any of Caml's imperative features in any part of your solution.) Give an example input for which this implementation is significantly faster than either solution in problem 1.

   (b) Prove that the translation you defined in 2(b) is correct. Warning: This is difficult; you should probably talk to Dan about how to set up the right induction hypothesis. We will also give credit for "half a proof," i.e., a proof that every terminal configuration in the source is possible in the target or vice-versa (as opposed to proving both).

What to turn in:

- Caml source code for 1d and 1g. (Rather than submit 1b just put a comment in the file explaining what you changed.)
- Hard-copy (written or typed) answers to all other problems.
- If you do the first extra credit, you can do it in `interp.ml` or another file. If the latter, submit a modified Makefile.

Email your source code to Erika as `firstname-lastname--interp.ml`. The code should untar/unzip into a directory called `firstname-lastname--hw2`. Hard copy solutions should be put in Erika's grad student mailbox, in the envelope outside her office, or given to her directly.